

The Mathematical Foundations for Mapping Policies to Network Devices

(Technical Report)

Dinesha Ranathunga¹, Matthew Roughan¹, Phil Kernick² and Nick Falkner³

¹*ACEMS, University of Adelaide, Adelaide, Australia*

²*CQR Consulting, Unley, Australia*

³*School of Computer Science, University of Adelaide, Adelaide, Australia*

{dinesha.ranathunga, matthew.roughan, nick.falkner}@adelaide.edu.au, phil.kernick@cqr.com

Keywords: Network-security, Zone-Conduit model, Security policy, Policy graph.

Abstract: A common requirement in policy specification languages is the ability to map policies to the underlying network devices. Doing so, in a provably correct way, is important in a security policy context, so administrators can be confident of the level of protection provided by the policies for their networks. Existing policy languages allow policy composition but lack formal semantics to allocate policy to network devices. Our research tackles this from first principles: we ask how network policies can be described at a high-level, independent of firewall-vendor and network minutiae. We identify the algebraic requirements of the policy-mapping process and propose semantic foundations to formally verify if a policy is implemented by the correct set of policy-arbiters. We show the value of our proposed algebras in maintaining concise network-device configurations by applying them to real-world networks.

1 INTRODUCTION

Managing modern-day networks is complex, tedious and error-prone. These networks are comprised of a wide variety of devices, from switches and routers to middle-boxes like firewalls, intrusion detection systems and load balancers. Configuring these heterogeneous devices with their potentially complex policies manually, box-by-box is impractical.

A better approach is a *top-down configuration* where device configurations are derived from a high-level user specification. Such specifications raise the level of abstraction of network policies above device and vendor-oriented APIs (e.g., Cisco CLI, OpenFlow). Doing so, provides a *single source of truth*, so, security administrators for instance, can easily determine who gets in and who doesn't (Howe, 1996).

In recent years, several research groups have proposed high-level policy specification languages for both Software Defined Networking (SDN) (Soulé et al., 2014; Reich et al., 2013; Foster et al., 2010; Prakash et al., 2015) and traditional networks (Bartal et al., 2004; Guttman and Herzog, 2005). A common requirement in these languages is the ability to map the abstract policies to the underlying physical network. Doing so accurately, is essential to:

- enforce policy correctly;
- track policy changes per network device, so, redundant policy updates can be avoided;

But, mapping policies to the physical network devices has challenges:

- policy needs to be *decoupled* from the network (i.e., free of vendor and network-centric minutiae). A policy shouldn't need to change when the device vendor changes, or every time IP address changes occur;
- the mapping must be *formally verifiable*. Precise and unambiguous mathematical semantics would eliminate wishful thinking pitfalls in deploying policies to networks. These semantics give, for instance, security administrators assurance that their intended policies are enforced by the correct firewalls, rendering the expected security outcome.
- policies can have complex semantics including node and link properties; and
- the mapping should be as efficient as possible.

The solution we propose provides a generic framework to map policies to network devices algebraically. We illustrate it's use by considering security policies, because incorrect deployment of these policies in domains such as Supervisory Control and Data Acquisition (SCADA) networks can result in catastrophic outcomes including the loss of human lives! However, the principles involved, can easily be extended to policies involving traffic measurement, QoS, load balancing and so on.

In developing our algebras, we have derived the properties and constraints of sequentially- and parallelly-composed policies for various policy contexts. These policy-composition semantics must be preserved, when mapping policy to devices, to ensure

correct deployment. Using an algebraic framework also makes the policy-mapping process efficient.

We will describe the implementation of our proposed algebras and demonstrate their use in deploying security policies to real SCADA networks. Particularly, we will show its value in maintaining a clear, concise set of firewall configurations. Moreover, our approach allows administrators to conduct “what if” analysis by changing policy and/or network topology and observe their effect on the network devices required to implement the changes.

2 BACKGROUND AND RELATED WORK

“The advantages of implicit definition over construction are roughly those of theft over honest toil.”

Bertrand Russel

The quote is salient because network installations are commonly built from *bottom-up*, *i.e.*, a network-device is purchased, and configurations written. The policy is the result of the configuration, which is the consequence of a purchasing decision. So, the policy is implicitly defined as a result of many small decisions that interact in complex ways. Instead, best-practice guides (Byres et al., 2005; ANSI/ISA-62443-1-1, 2007) suggest designing the policy first, and only then determining how to implement it.

Solutions that employ such a *top-down* network configuration approach have been proposed in both SDN (Soulé et al., 2014; Anderson et al., 2014; Prakash et al., 2015) and traditional networks (Bartal et al., 2004; Cisco Systems, 2014). They allow creation and maintenance of a *single high-level network-wide policy* (*i.e.*, source of truth), so network administrators can easily determine, for instance, who gets in and who doesn’t. These high-level policies should be decoupled from network and device-vendor intricacies, so they capture the *policy intent* and not the *policy implementation*.

Capturing intent has several benefits: it allows policy to be distinguished from network to assist with change management; it enables accurate comparison of organisational policies to industry best-practices to evaluate compliance; and it allows policy semantics to be expressed without network intricacies like IP addresses. But, most research towards high-level policy languages (Bartal et al., 2004; Cisco Systems, 2014; Soulé et al., 2014), still requires hostnames and/or IP addresses to be explicitly input to the policy specification, to implement policy on a network instance.

If the high-level policy definition is built on *formal mathematical constructions*, then there are no implicit properties and it provides a truly sound foundation

for everything that follows. The formalism would allow construction of complex and flexible policies and support reasoning about the policies. For instance, we could precisely compare a defined policy with industry best-practices in (ANSI/ISA-62443-1-1, 2007; Byres et al., 2005) for compliance and reduce network vulnerability to cyber attacks (Anonymous, 2016).

It is equally important to map policy to devices using a formal approach. For instance, we can be confident of the blanket of protection provided for our network *if and only if* we could prove that an intended security policy is implemented by the correct set of network firewalls. Existing top-down configuration languages (Bartal et al., 2004; Prakash et al., 2015; Cisco Systems, 2014) allow creation of network-wide high-level policies, but lack means to allocate policy to network devices in provably correct way.

In our related work (Anonymous, 2015), we have developed a high-level security policy specification that is network and device-vendor independent to perform *top-down* configuration of network firewalls. The language semantics allow these high-level policies to be easily understood by humans. In this paper, we investigate the underlying requirements for mapping high-level policy to network devices.

3 ABSTRACT POLICIES

Abstractions are key to constructing high-level policies. A good policy abstraction should capture the underlying concepts naturally and concisely. For instance, a policy may be arbitrated using one or more network devices. A good abstraction should decouple *what* is arbitrated from *how* it is arbitrated.

A simple abstraction that is commonly used to decouple policy from the network is (*endpoint-group, edge*) (Anonymous, 2015; Bartal et al., 2004; Soulé et al., 2014; Prakash et al., 2015). An *endpoint* is a grouping of elements based on common physical or logical properties- *e.g.*, a subnet, a user-group, a collection of servers, *etc.* An *endpoint* is also the smallest unit of abstraction a policy can be applied to. An *edge* specifies the relationship between the endpoints, *i.e.*, it describes what communication is allowed between the endpoints. An edge typically consists of a Boolean predicate (*i.e.*, classifier) that matches traffic packet header details to perform some action. An edge from endpoint *S1* to *S2* with predicate *R* means that a correct implementation should act in some specified way toward traffic from all members (*e.g.*, hosts) in *S1* to those in *S2* satisfying *R*.

Consider the example network in Figure 1(a), our high-level policy might be – “we want to measure all HTTPS flows from *S1* to *S4*”. This intent can be expressed using the (*endpoint-group, edge*) abstraction

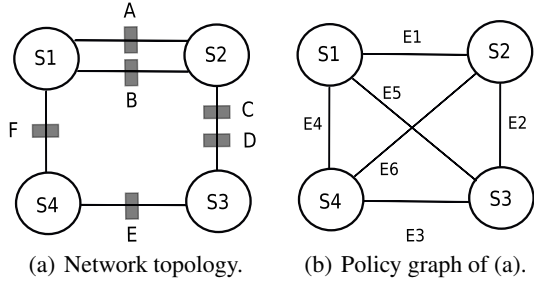


Figure 1: Example network consisting of four endpoints ($S1 - S4$) and six policy arbiters A, B, \dots, F (a). The corresponding policy graph is shown in (b) with these endpoints and policy edges ($E1, E2, \dots, E6$) between them.

as $p := S1 \rightarrow S4 : \text{collect https}$, where $p \in \mathcal{P}$ is an element of the possible space \mathcal{P} of policies.

We now want to map the policy to enforcing devices (*i.e.*, arbiters). It looks naively easy. Just implement the policy on arbiter F . However, the problem is in general more complicated. For example, consider policy from $S1$ to $S2$. If arbiter A captures flows described by policy p_A and likewise p_B , then the combination is $p_{A \cap B}$ because the policy expresses that flows be collected but load balancing across the two paths could result in either being used by a given packet.

For another example, consider policy from $S2$ to $S3$. If arbiters C and D capture flows described by policy p_C and p_D respectively, then the combination would yield their *union* because both arbiters in the path are used for flow collection. So, when mapping policies to network devices, they must be composed in a way that preserve the topology-specific semantics.

We will, in general, denote the policy composition resulting from parallel routes as $p_A \oplus p_B$ and that resulting from serial routes as $p_A \otimes p_B$. In this example, $p_A \oplus p_B = p_{A \cap B}$ and $p_C \otimes p_D = p_{C \cup D}$. Note though, the meaning of \oplus, \otimes are policy-context dependent.

The problem of mapping our high-level policy p (from $S1$ to $S4$) to network devices, may even be more complicated than simply implementing the policy on F . What happens if F drops out? Surely we would still want to collect HTTPS flows that traverse the redundant paths from $S1$ and $S4$ (*e.g.*, path via $S2$ and $S3$). So, p must also be mapped to the policy arbiters along the path $S1 - S2 - S3 - S4$, to cater for this redundancy. But then, should we map the policy to all of these arbiters or a subset of them?

Given the parallel routes between $S1$ and $S2$ it is easy to see that we need to map p to both A and B to preserve the semantics of \oplus . But, when arbiters are in series (*e.g.*, C, D) we have the choice of mapping policy to both or just one to preserve the semantics of \otimes . Mapping to both may be unnecessary and inefficient and arbiters may have limits on their capabilities.

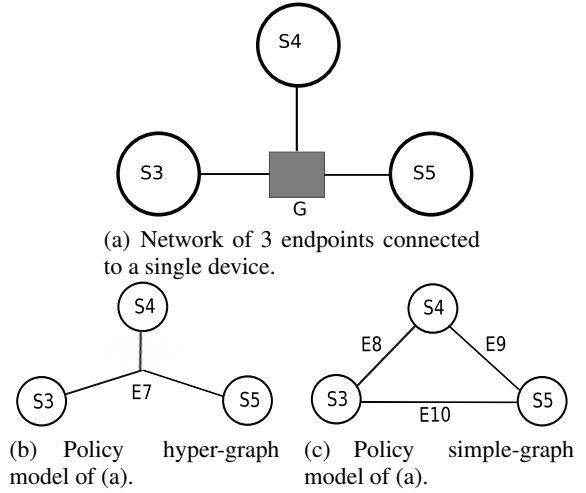


Figure 2: Policy-graph definition alternatives.

The policy arbiters may not always be in series or parallel. For instance, consider the star-topology involving the endpoints $S3, S4$ and $S5$ in Figure 2(a). The question of how to map policy to topology gets interesting in this case. We could loosely map policy to arbiter G using a *hyper-edge* $E7$ (Figure 2(b)). But, doing so adds to the complexity of the policy-graph and decreases its precision. For instance, it allows a $1 : n$ relationship between hyper-edges and policies, so, we must track the policy of *every endpoint-pair* in a hyper-edge to correctly map its policy to network devices. Also, a hyper-edge could potentially map to a large portion of the network (*e.g.*, when interconnected to many endpoints), so, its policy can be complex, difficult to understand and debug.

We can overcome these problems by using a simple-edge mapping instead (Figure 2(c)). By doing so, we also gain significant benefits. For one, the policy graph is simplified and its precision is increased—the $1 : 1$ relationship between edges and policies now mean we only need to track a single policy for each edge, to map policy accurately to devices. For another, our simplified abstraction (*endpoint-pair, edge*) behaves as a network-slice (Gutz et al., 2012)—a piece of the network that can be programmed independently from the rest of the network — so, debugging policy errors is easy. Moreover, users from distinct policy subdomains (*e.g.*, Engineering, Corporate Administration) can manage the network-slices (*i.e.*, policy-edges) applicable to their domain.

We may also need to consider paths that consist of one more intermediate endpoints, when implementing policy between an endpoint pair. For instance, consider implementing a security policy $f := S1 \rightarrow S3 : \text{http}$ in the network in Figure 1(a). Once im-

plemented on the arbiters (*i.e.*, firewalls), the policy should allow HTTP traffic flow from $S1$ to $S3$. But, the policy can only be deployed correctly if endpoints $S2, S4$ can route or forward HTTP traffic.

This ability of an endpoint to route or forward traffic can be restrictive. For instance, in a security policy context, an endpoint can group systems (*e.g.*, hosts) with similar security requirements (ANSI/ISA-62443-1-1, 2007). So, high-risk systems can be grouped in to one endpoint and only *safe* traffic that originate and/or terminate at the endpoint is permitted by the security policy. Enabling this endpoint to transit-traffic could potentially expose the high-risk systems within to cyber attacks. So, an endpoint's *traffic transitivity* capability must be considered in constructing valid device-paths between endpoints and captured explicitly in the mapping process.

The intent of our high-level policy p is to collect HTTPS flow data from $S1$ to $S4$. Similarly, traffic measurement policies can be defined between any pair of endpoints in the network, so the corresponding (endpoint-pair, edge) tuples collectively construct a *policy-graph* (Figure 1(b)). Each (logical) edge in this graph maps to a physical network path comprising of policy arbiters and zero or more intermediate endpoints. For instance, policy-edge $E4$ in Figure 1(b) implements our policy p .

So far, we have described several key requirements of a policy-to-device mapping process. We next illustrate these using more detailed examples involving a typical set of policies found in practice. The examples also show the use of our (*endpoint-pair, edge*) abstraction in high-level policy description.

3.1 Quality of Service (QoS) Policies

QoS policies can be used, for instance, to provide bandwidth guarantees for certain types of traffic. The policy arbiters in this context, would commonly be QoS capable routers or switches. Imagine we need to provision a minimum bandwidth of 100MB/s for HTTP traffic flow from $S1$ to $S4$ in Figure 1(a). The high-level policy intent can be expressed using the endpoint-pair $S1, S4$ and an edge ($E4$) between them: $\min(S1 \rightarrow S4 : \text{http}, 100\text{MB/s})$. In reality, similar QoS policies between any endpoint-pair can be expressed using the same policy-graph in Figure 1(b).

Parallel and serial (QoS-device) topologies also have an impact on the end QoS policy. Consider the parallel topology between $S1$ and $S2$ in Figure 1(a), if we assume p_A and p_B provide bandwidth guarantees of $B1$ and $B2$ respectively, then with load balancing, the resultant QoS policy (p_R) can provide a total bandwidth guarantee $p_R = p_A \oplus p_B = \text{sum}(B1, B2)$.

When the devices are in series (*e.g.*, topology between $S2$ and $S3$ in Figure 1(a)), the resultant policy

provides a bandwidth guarantee given by $p_C \otimes p_D = \min(B3, B4)$ where $B3, B4$ are the bandwidth guarantees provided by p_C and p_D .

This example highlights how policies can be composed using the generic semantics \oplus and \otimes in different policy contexts. The actual meaning of these operators is policy-type dependant. For instance, in the traffic measurement policy example, \otimes represented *union* while here it has meaning of *minimum*.

We can see that endpoint traffic-transitivity also impacts the ability to implement a QoS policy correctly in the network. For instance, consider policy $\min(S1 \rightarrow S3 : \text{http}, 50\text{MB/s})$, the required bandwidth guarantee can only be provided if $S2, S4$ routes or forwards traffic to $S3$.

3.2 Security Policies

The security policies we consider here are access-control policies in a network. The policy arbiters in this context, would be traffic filtering devices (*e.g.*, firewalls, SDN switches). The endpoints could be zones or user groups. Imagine we want to enable only SSH traffic from $S1$ to $S4$ (Figure 1(a)). The high-level policy can be expressed using the endpoint-pair $S1, S4$ and an edge between them as $S1 \rightarrow S4 : \text{ssh}$ with an implicit *deny-all* in place.

When the traffic filtering devices are in parallel (*e.g.*, topology between $S1$ and $S2$ in Figure 1(a)), the resultant security policy (p_R), has meaning- *all packets that can possibly be allowed through* – and is the *union* of the packet sets allowed through by the individual devices. We take union conservatively because intrusions and attacks are usually carried out through permitted traffic. So, if p_A and p_B allows packet sets Q and T respectively, then $p_R = p_A \oplus p_B = p_{Q \cup T}$.

When the devices are in series (*e.g.*, topology between $S2$ and $S3$), the resultant policy permits the *intersection* of the packet sets V, W allowed by the policies of C, D respectively -*i.e.*, $p_R = p_C \otimes p_D = p_{V \cap W}$.

With security policies, it is often useful to have endpoints that group systems with similar security requirements. Doing so, allows to define a *single policy* for all members of an endpoint – a clean and concise abstraction of network security. But, a generic (endpoint-pair, edge) abstraction cannot capture such network-security specific policy concepts precisely.

So, we need concrete definitions for what an endpoint and an edge means for each policy context to capture policy-specific intricacies. We will show later how to define these concretely for security policies.

3.3 Traffic Measurement Policies

Traffic measurement policies can be used to implement, for instance, NetFlow (v9) on the example network shown in Figure 1(a). The policy arbiters

A, B, \dots, F here, would be NetFlow capable devices (usually routers or switches). The endpoints could be subnets or zones in the network.

When we considered the parallel routes between $S1$ and $S2$, the resultant policy (p_R) with meaning—*the flows that are guaranteed to be captured by the topology (without sampling)*, constitutes of the *intersection* of the packet sets of A and B . So, if p_A and p_B capture packet sets Q and T respectively, then $p_R = p_A \oplus p_B = p_{Q \cap T}$.

We also showed that when the devices are in series (e.g., topology between $S2$ and $S3$ in Figure 1(a)), the resultant policy captures the *union* of the packet sets V, W captured by the policy of the devices C and D respectively, i.e., $p_R = p_C \otimes p_D = p_{V \cup W}$.

Policy Mapping Vs Routing: We can see that in mapping policies to network devices, our discussion above relates to network paths or routes. This is no accident. The problem we aim to solve has many parallels with routing. But here,

- we often look for all paths, not just best paths.
- we do not need to employ decentralised protocols. A logically-centralised implementation can offer a robust and efficient solution as demonstrated in SDN (Anderson et al., 2014; Foster et al., 2010; Soulé et al., 2014; Prakash et al., 2015).

The aim in routing is to determine the path that optimises a given path-metric (e.g., shortest-path routing finds a path between endpoints with a minimum distance). Our target problem is different: we need to determine the set of arbiters in a network a given policy should be implemented on. So, constructing all feasible paths is crucial because, for instance, a security policy between two endpoints can only be correctly implemented (e.g., to block all Telnet traffic) if all redundant paths between them are taken into account.

We also expect a relatively low number of endpoints (e.g., in §6 we will see this number is typically < 25 in a SCADA network), through network grouping. As we will show in §5, a low endpoint-count makes our policy-mapping algorithm computationally tractable. So, a logically-centralised implementation can be considered. But, in routing, the number of endpoints (e.g., individual gateways) can be high for a large network, so, distributed means to computing a solution is essential (requiring de-centralised protocols like OSPF, BGP). Our policy-mapping algorithm can also be implemented distributedly if necessary.

Algebras have also been proposed in meta-routing (Dynerowicz and Griffin, 2013), to provably choose paths that optimise various path-metrics. But, these algebras *do not support specification of node properties* such as traffic transitivity. Specification of link

properties is possible and we could translate node properties to link properties, but, the workaround is not elegant. A better approach is to have an algebra that capture node properties explicitly.

We described in §3.2, the need to define endpoints and edges concretely for each policy context. By doing so, we can incorporate the intricacies associated with each policy type. We illustrate the idea next using security policies and their concretised (endpoint-pair, edge) abstraction—*the Zone-Conduit model*.

The Zone-Conduit Model: Lack of internal-network segmentation is a key contributor to the fast propagation of threats and attacks in a network (Byres et al., 2005). So, ANSI/ISA have proposed the *Zone-Conduit model* as a way of segmenting and isolating the various sub-systems in a SCADA network (ANSI/ISA-62443-1-1, 2007).

A *zone* is a logical or physical grouping of an organisation’s systems with similar security requirements so that a *single policy* can be defined for all zone members. A *conduit* provides the secure communication path between two zones, enforcing the policy between them (ANSI/ISA-62443-1-1, 2007). A conduit could consist of multiple links and firewalls but, logically, is a single connector.

It is easy to see that the Zone-Conduit model is a concrete instance of the (endpoint-pair, edge) abstraction for high-level security policy specification. The zones and conduits are concrete definitions of endpoints and edges. In defining them, important network-security characteristics such as a *single zone-policy*, are captured concisely.

The Zone-Conduit model abstracts *how* security policy is enforced, so users can focus on *what* policy to enforce. The model intuitively decouples policy from topology, so, high-level security policies can be described free of network and vendor intricacies.

The ISA Zone-Conduit model in its original description lacks precision for policy specification. We use the extensions proposed in (Ranathunga et al., 2015b) to increase its precision, e.g., we add Firewall-Zones to specify firewall-management policies.

Zone-Conduit Policies: A conduit policy is essentially an ordered set of rules $[p_1, p_2, \dots, p_n]$ that act on packet sequences to accept, deny, or in some cases, modify them. In our related work (Anonymous, 2015; Anonymous, 2016), we have identified properties and constraint required in a Zone-Conduit based policy description to make these rules vendor and device (i.e., implementation) independent and have rule-order independent semantics.

To summarise, we adopt a security whitelisting model, i.e., we restrict policies to express positive

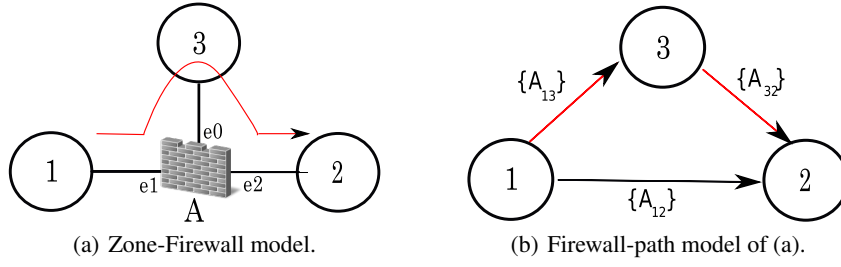


Figure 3: Example of traffic packets traversing a single firewall interface twice. As shown by the path in red, if traffic from zone 1 to 2 is allowed to transit zone 3, firewall interface $e0$ is traversed twice. In this situation, the direct path from zone 1 to 2 is more efficient. So, we deem firewall path $A_{13}A_{32}$ invalid (i.e., $=\emptyset$), since $h(A_{13}) = h(A_{32}) = A$, where $h : F \rightarrow W$.

abilities¹ and deny all inter-zone flows that are not explicitly allowed (Anonymous, 2015). Doing so, renders the rule order irrelevant in a policy, allowing the policy to be converted consistently to different vendor firewalls. So, the underlying vendor can change without requiring policy alterations. Policy makers can also add or remove policy rules oblivious to the ordering of the rules. By being explicit, we also prevent services being accidentally enabled implicitly.

Directed- vs Undirected-Conduit Policy: The policy on an undirected-conduit can be expressed using two directed-conduit policies. Directed-conduits are important in understanding how policy should be implemented on device interfaces. For instance, consider implementing a security policy $p := Z1 \rightarrow Z2 : https$ on the network shown in Figure 3(a). An undirected-conduit can only map the policy to firewall interfaces $e1, e2$. But, to implement the policy correctly, we additionally need to know if it should be implemented *inbound* or *outbound* on these interfaces. A directed-conduit provides this directionality.

However, analysis using directed-conduits can also lead to problems. For instance, it seems feasible to implement our policy p along the path highlighted in red in Figure 3(b). But further inspection of Figure 3(a), reveals that traffic packets traversing this path would require to traverse firewall interface $e0$ twice.

We invalidate such directed-conduit paths via a mapping $h : F \rightarrow W$ where $F = \{\text{directed firewalls}\}$ and $W = \{\text{firewalls}\}$. A directed-firewall in the Zone-Conduit graph $G = (Z, C)$ is defined as

Definition 1 (Directed firewall). A directed firewall t_{ij} is a firewall $t \in W$ that filters traffic on directed-conduit $(i, j) \in C$.

Then we can check if the directed-firewalls in a path map to the same physical firewall (i.e., $h(A_{13}) = h(A_{31}) = A$) and deem that path invalid.

¹Refers to the ability to initiate or accept a traffic service.

4 MAPPING ALGEBRA

We now outline our proposed algebras to map high-level policy to network devices. We start by making the distinction between *primary* and *secondary* policy-edges, and then later show how the latter can be derived algebraically using the former.

4.1 Firewall-path Construction

As we described earlier, a policy-edge enforces policy between two endpoints in a network. A policy-edge can be classified as a *primary*- or a *secondary*-edge depending what arbiters are implemented within. A primary-edge can enforce policy *only* using policy arbiters. A secondary-edge enforces policy using both arbiters and one or more additional endpoints.

We demonstrate the idea using security policies and the simple Zone-Conduit graph $G = (Z, C)$ in Figure 4(a). The firewall composition of each primary-conduit in the model is shown (Figure 4(b)). The example secondary-conduit C_{13} filters traffic flow from zone 1 to 3, using several directed-firewalls and transit zones 2, 4. The set of all directed-conduits are given by $DC = \{C_{ij} \mid (i, j) \in C\}$.

A policy-graph is essentially an automation that moves traffic packets from one endpoint to another using policy-edges. So, regular expressions (the natural language of finite automata), can capture the packet-processing behaviour of this model; a path encoding is a concatenation of directed-devices (i.e., policy-arbitration steps pq) and a set of paths is encoded as a union of paths. Past work (Anderson et al., 2014) has shown, all single-path encodings stem from the Kleene star operator ($*$) on the set of directed-devices.

In a security policy context, the automation means that a single firewall-path encoding is a concatenation of directed-firewalls. Each path depicts a sequence of traffic filtering steps and is an element of F^* .

But, the Zone-Conduit model is a logical representation, so, every firewall-path encoding in F^* may not be valid. We follow the design principles below to filter-out invalid paths:

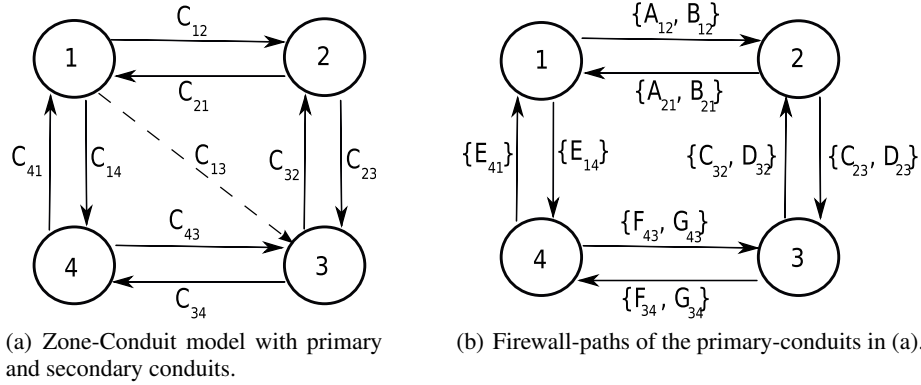


Figure 4: Zone-Conduit model depicting primary-conduits C_{ij} and a secondary-conduit C_{13} enabled by the transit zones 2 and 4 shown in (a). The firewall-paths of the primary-conduits are shown in (b).

- non-elementary paths in the Zone-Conduit model must be *excluded*.
- when traversing elementary Zone-Conduit paths, traffic packets shouldn't traverse a particular device interface more than once. So, for directed-firewalls $t_{ij}, t_{jk} \in F$ the concatenation $t_{ij}t_{jk}$ is valid iff $h(t_{ij}) \neq h(t_{jk})$. We demonstrated the idea using the simple example in Figure 3 earlier.
- traffic cannot flow through a non-transit zone. Typical zones this restriction applies to are the protected SCADA-Zones and Firewall-Zones.

We have outlined the design principles for constructing a single firewall-path in a network. In actuality, these principles and methods apply equally when constructing device-paths in other policy contexts.

We define single firewall-path concatenation based on the above design principles

Definition 2 (Single firewall-path concatenation). Take $a \in F^*$ s.t. $a = t_{d_1d_2}t_{d_2d_3}t_{d_3d_4}\dots t_{d_nd_{n+1}}$ where $t_{d_id_j} \in F$. Also take $b \in F^*$ s.t. $b = s_{g_1g_2}s_{g_2g_3}s_{g_3g_4}\dots s_{g_mg_{m+1}}$ where $s_{g_ig_j} \in F$. Then firewall-path concatenation from $F^* \times F^* \rightarrow F^*$ is

$$ab = \begin{cases} t_{d_1d_2}\dots t_{d_nd_{n+1}}s_{g_1g_2}\dots s_{g_mg_{m+1}} & \text{if } d_{n+1} = g_1 \\ \text{and } g_i \neq d_j; \forall i, j, i > 1 \\ \text{and } h(t_{d_id_j}) \neq h(s_{g_kg_l}); \forall i, j, k, l \\ \phi, & \text{otherwise.} \end{cases}$$

and $a\phi = \phi a = \phi; \forall a \in F^*$

Concatenation in Definition 3 is a binary operation that constructs only elementary firewall-paths.

Consider two directed-firewalls X, Y with policies p_X and p_Y that accepts packet sets R and T . The resultant policy of the concatenated firewall-path XY has an action similar to that of sequential firewalls and can be denoted as $(p_X \otimes p_Y)(s) = p_{R \cap T}(s)$ where s is a packet sequence. The resultant policy action also depends on the policy context,. For instance, with

traffic measurement policies, the outcome is similar to that of serial arbiters- i.e., $p_X \otimes p_Y = p_{R \cup T}$.

We define a set of directed-firewall paths as a union of elements in F^* . Again, consider our directed-firewalls X, Y from before. If these described two distinct paths, then the resultant policy of the path union $X \cup Y$ is that of parallel firewalls, i.e., $(p_X \oplus p_Y)(s) = p_{R \cup T}(s)$ where s is a packet sequence. This result is also context dependent, so, for traffic measurement policies, the outcome is similar to that of parallel arbiters, i.e., $p_X \oplus p_Y = p_{R \cap T}$.

We also extend concatenation in Definition 2 to $S = \{\text{Power-set of } F^*\}$

Definition 3 (Multiple firewall-path concatenation). Take $a, b \in S$ s.t. $a = \{a_0, a_1, \dots, a_x\}, b = \{b_0, b_1, \dots, b_y\}$ where $a_i, b_j \in F^*$. Then firewall-path concatenation from $S \times S \rightarrow S$ is given by

$$ab = \{a_ib_j\}; \forall i, j$$

Then, Definition 3 allows all possible sets of firewall paths to be constructed from the union of elements in S (i.e., $\forall a, b \in S, a \cup b \in S$). Then $(S, \cup, \cdot, \hat{0}, \hat{1})$ is an idempotent semiring with

$\hat{0} = \phi$; empty set; and

$\hat{1} = \{\epsilon\}$; empty-string set where ϵ is the identity element of the concatenation operation.

It is important to note here that the properties of the operators \cup and \cdot actually dictate rules for firewall-path construction. So, when valid firewall-paths are built, they must be composed in a way that preserves these semantics. For instance, \cup is commutative while \cdot is not. So, the order of the directed-firewalls matter, when constructing a single firewall-path, but, are irrelevant when constructing multiple paths.

Similarly, we can construct single and multiple device-paths for other policy contexts and obtain the semiring result for $(S, \cup, \cdot, \hat{0}, \hat{1})$ per security policies.

4.2 Mapping Policy to Arbiters

In the previous section we described how the semiring $(S, \cup, \cdot, \hat{0}, \hat{1})$ constructs sets of device-paths between policy-graph endpoints. The sequential (*i.e.*, \otimes) and parallel (*i.e.*, \oplus) policy-composition operators in § 3 can now be used to construct the policies of these paths. For instance, assume we have to implement a high-level policy p_{ij} on arbiters q_{kl} that lie in the paths from i to j . All applicable device-paths from i to j can be constructed as per § 4.1 and given by $S_{ij} = \{q_{a_1a_2}q_{a_2a_3}\dots q_{a_{n-1}a_n}, \dots, q_{b_1b_2}q_{b_2b_3}\dots q_{b_{m-1}a_m}\}$. Then, the high-level policy p'_{ij} derived from the individual arbiter policies p'_{kl} is

$$\begin{aligned}
p'_{ij} &= (p'_{a_1 a_2} \otimes p'_{a_2 a_3} \cdots \otimes p'_{a_{n-1} a_n}) \\
&\oplus (\dots\dots\dots) \\
&\oplus (p'_{b_1 b_2} \otimes p'_{b_2 b_3} \cdots \otimes p'_{b_{m-1} b_m}). \quad (1)
\end{aligned}$$

Mapping the intended high-level policy p_{ij} to the arbiters is now a matter of finding p'_{ij} for all arbiters such that $p'_{ij} = p_{ij}$. But deriving such a mapping is non trivial because the meaning of \oplus and \otimes depends on the policy context. For instance, with security policies \oplus means *union* and \otimes means *intersection*. So, a simple solution that supports *defence in depth* would be to implement the access-control policy p_{ij} on every sequential firewall across all paths.

For another instance, \oplus means *summation* and \otimes means *minimum* in QoS policies. So, a required bandwidth guarantee p_{ij} can be split across multiple paths with sequential arbiters in each path guaranteeing only a portion of the total bandwidth.

Similarly, with traffic measurement policies, \oplus means *intersection* and \otimes means *union*. So, each path must implement policy p_{ij} . This can be achieved in multiple ways: (i) we could implement p_{ij} on every sequential arbiter of a path for redundancy; or (ii) efficiently have only one arbiter per path implementing the policy; or (iii) find an intermediate solution by partitioning p_{ij} between sequential arbiters in a path.

Irrespective of the policy context, the underlying requirement when mapping policy to network devices is to adhere to the semantics of (1).

4.3 Computation of All Firewall-paths

We have managed to reduce the policy-to-device mapping problem to the semantics of (1) in the previous section. We now describe how an algorithm can be developed to compute the device-paths of (1) efficiently. Again, we demonstrate the idea using security policies and the Zone-Conduit model.

We can represent the firewall-paths of the primary-conduits using a *generalised Adjacency matrix* A . Here, $A(i, j)$ is the firewall-path of primary

conduit $C_{ij} \in DC$. For our example in Figure 4(b),
 $A =$

$$\left[\begin{array}{cccc} Z_1 & Z_2 & Z_3 & Z_4 \\ \{\epsilon\} & \{A_{12}, B_{12}\} & \phi & \{E_{14}\} \\ \{A_{21}, B_{21}\} & \{\epsilon\} & \{C_{23}, D_{23}\} & \phi \\ \phi & \{C_{32}, D_{32}\} & \{\epsilon\} & \{F_{34}, G_{34}\} \\ \{E_{41}\} & \phi & \{F_{43}, G_{43}\} & \{\epsilon\} \end{array} \right] \quad (2)$$

Given such a matrix A , the solution to the problem of finding the set of all *valid* firewall-paths between zones is a matrix A^* such that

$$A^*(i, j) = \{\text{valid primary- and secondary-conduit firewall-paths from zone } i \text{ to } j\} \quad (3)$$

We developed the following right iteration algorithm² to compute A^* , inspired by algorithms in meta-routing (Dynerowicz and Griffin, 2013). In doing so, we allow the node property: *traffic transitivity*, to be explicitly specified via a *zone-transitivity* matrix T .

$$\begin{aligned} A^{<0>} &= I \\ A^{<k+1>} &= (A^{<k>}T \cup I)A \end{aligned} \quad (4)$$

The *zone-transitivity* matrix T is defined as

$$T(i, j) = \begin{cases} \{\epsilon\} & \text{if } i = j \text{ and } \text{zone_transitivity}(i) = 1 \\ \emptyset, & \text{otherwise.} \end{cases} \quad (5)$$

and I is the multiplicative-identity matrix

$$I(i, j) = \begin{cases} \hat{1} & \text{if } i = j \\ \hat{0}, & \text{otherwise.} \end{cases} \quad (6)$$

For bounded semirings we only iterate $n - 1$ times to converge to A^* , where n is the number of nodes in the Zone-Conduit model, *i.e.*,

$$A^* = A^{<n-1>}$$

We have defined T , A^* and the right-iteration algorithm for a security-policy context, but these can equally be defined for other policy contexts.

If we apply our algorithm in (4) to the example in Figure 4(b), $n = 4$, so, $A^* = A^{<3>}$ and let's assume for simplicity that all zones are transitive. Then

$$T = \begin{bmatrix} \{\mathbf{\varepsilon}\} & \phi & \phi & \phi \\ \phi & \{\mathbf{\varepsilon}\} & \phi & \phi \\ \phi & \phi & \{\mathbf{\varepsilon}\} & \phi \\ \phi & \phi & \phi & \{\mathbf{\varepsilon}\} \end{bmatrix} \quad (7)$$

and we see that $I = T$ in this instance.

$$\begin{aligned} A^{<1>} &= (A^{<0>}T \cup I)A = (IT \cup I)A \\ &= (I \cup I)A = A \\ A^{<2>} &= (AT \cup I)A \end{aligned}$$

²See appendix for proof.

So, $A^* = A^{<3>} = (A^{<2>}T \cup I)A =$

$$\begin{bmatrix} \{\epsilon\} & \eta & \kappa & \theta \\ \mu & \{\epsilon\} & \nu & \beta \\ \gamma & \lambda & \{\epsilon\} & \rho \\ \xi & \delta & \zeta & \{\epsilon\} \end{bmatrix} \quad (8)$$

where for instance

$$\kappa = \{A_{12}C_{23}, A_{12}D_{23}, B_{12}C_{23}, B_{12}D_{23}, E_{14}F_{43}, E_{14}G_{43}\} \quad (9)$$

All valid firewall-paths from zone 1 to 3 are in κ .

Let's now assume that zone 4 is non-transitive, then $zone.transitivity(4) = 0$, and by (5),

$$T = \begin{bmatrix} \{\epsilon\} & \phi & \phi & \phi \\ \phi & \{\epsilon\} & \phi & \phi \\ \phi & \phi & \{\epsilon\} & \phi \\ \phi & \phi & \phi & \phi \end{bmatrix} \quad (10)$$

We re-calculate $A^* =$

$$\begin{bmatrix} \{\epsilon\} & \{A_{12}, B_{12}\} & \eta & \theta \\ \{A_{21}, B_{21}\} & \{\epsilon\} & \{C_{23}, D_{23}\} & \mu \\ \gamma & \{C_{32}, D_{32}\} & \{\epsilon\} & \rho \\ \xi & \delta & \zeta & \{\epsilon\} \end{bmatrix} \quad (11)$$

where for instance

$$\eta = \{A_{12}C_{23}, A_{12}D_{23}, B_{12}C_{23}, B_{12}D_{23}\} \quad (12)$$

The updated firewall-paths from 1 to 3 are in η . In comparison to (9), we see that the paths via zone 4 (i.e., $E_{14}F_{43}, E_{14}G_{43}$) have now been removed as the zone is no longer transitive. A^* can similarly be calculated for other policy contexts to determine all valid device-paths between endpoint pairs.

We will next describe our implementation of the algorithm in (4). The implementation allows us to use these algebras to map policy to real network devices.

5 IMPLEMENTATION

Our policy mapping system is depicted in Figure 5, and we outline its details below. The system is currently implemented in Python, and allows mapping of high-level policies written in our own policy specification language, to network devices. We will make the system open source in the near future.

High-level security policy: The topology-independent policy input file created using our high-level policy specification language.

Network topology: The input network topology described in the XML-based graph file format *GraphML* (Graph Steering Committee, 2003). The file holds information of all devices in the network and their interconnections. The crucial aspects are the details of the topology near the policy arbiters (e.g., firewalls).

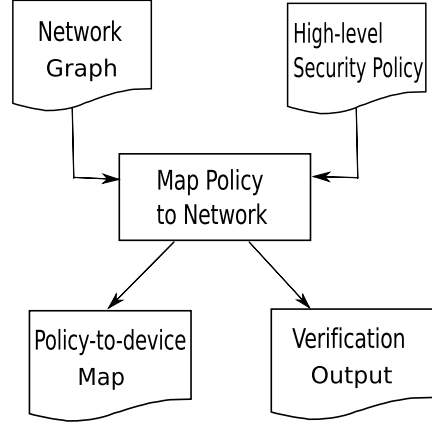


Figure 5: Policy to network-device mapping process.

Map policy to network: Compiles high-level policy to an Intermediate-Level (IL), generates the policy graph (e.g., Zone-Conduit model) of the input network and computes A^* as per §4.3 to map high-level policy to the devices (e.g., firewalls) in the network.

Policy-to-device map: The primary output depicting policy breakdown by device, interface and direction.

Verification output: Secondary output specifically suited for verification (e.g., policy errors).

Our system implements a high-level policy on a network by coupling the policy to the topology instance. Details of the coupling steps are discussed in detail in our previous work in (Anonymous, 2015). We outline here, the steps taken to map policy to the network devices. Again, we illustrate our implementation using security policies and the Zone-Conduit model, but the system can likewise be used in other policy contexts.

5.1 Zone-transitivity Matrix and Adjacency Matrix Construction

Our system first builds the *Zone-Firewall model*, containing the disjoint zones and their firewall interconnections using the input network topology. Additional Firewall-Zones, Abstract-Zones and Carrier-Zones (Ranathunga et al., 2015b) are added to the model as required. The primary conduits are then defined to create the Zone-Conduit model.

Next, the zone-transitivity capabilities are collated by compiling the high-level policy to IL policy. Then the $n \times n$ zone-transitivity matrix (T) is constructed (n is the number of zones in the network).

Implicit mappings between the primary conduits and their firewall compositions are automatically created when generating the Zone-Conduit model. These mappings are leveraged to construct the adjacency matrix A described in §4.3.

Algorithm 1 Right-iteration algorithm to compute A^* . The additive and multiplicative operators of Semiring S are \cup, \cdot (§ 4.1). The additive and multiplicative identities of S are $\hat{0}, \hat{1}$. The multiplicative-identity matrix and the zone-transitivity matrix are I and T . The set of zones in the network is Z .

```

1: procedure RIGHTITERATION( $(\cup, \cdot, \hat{0}, \hat{1}, A)$ )
2:    $R_0 \leftarrow I$ 
3:    $k \leftarrow 1$ 
4:   while  $k \leq |Z| - 1$  do
5:     for each  $i \in Z$  do
6:       for each  $j \in Z$  do
7:         if  $i=j$  then
8:            $R_{k+1}(i, j) \leftarrow I(i, j)$ 
9:         else
10:           $R_{k+1}(i, j) \leftarrow \hat{0}$ 
11:          for each  $q \in Z$  do
12:             $s \leftarrow R_k(i, q) \cdot T(q, j)$ 
13:             $R_{k+1}(i, j) \leftarrow R_{k+1}(i, j) \cup s$ 
14:             $R_{k+1}(i, j) \leftarrow R_{k+1}(i, j) \cup I(i, q)$ 
15:             $R_{k+1}(i, j) \leftarrow R_{k+1}(i, j) \cdot A(q, j)$ 
16:           $k \leftarrow k+1$ 

```

5.2 A^* Calculation

We compute A^* using the right iteration algorithm in (4). Our implementation is given below (Algorithm 1). We consider a centralised implementation (given typically low n), but, it can also be distributed across multiple nodes performing parallel computations.

Lines 2,3,10 initialise R_0 , k and $R_{k+1}(i, j)$ while lines 5,6,11 range over the set of zones Z in the network. At each step k , we store the result A_{k+1} in R_{k+1} .

We can see that the above algorithm has time complexity $O(n^4)$. This value suggests that the algorithm performance decreases exponentially as the number of zones in the network (n) increases. So, specifying policy per individual host (*i.e.*, zone size=1 so, large n) in *top-down* configuration makes policy mapping *extremely inefficient*. It proves more efficient to create network groups (*e.g.*, by subnet) and specify policy between them, so, a reasonably low value can be maintained for n . For instance, we will see in § 6 that in a SCADA network typically $n < 25$. Our mapping algorithm runs on average in 53 seconds on a standard desktop computer for each of these networks.

In calculating A^* , we considered all valid paths between two zones (hence time complexity of $O(n^4)$). The decision allows us to permit or block traffic along all possible communication paths between two zones, providing redundancy and *defence-in-depth* in the network. We also map policy uniformly to every firewall along a single-path, further boosting defence-in-depth. These decisions collectively create a more robust defence against cyber attacks.

However, in other policy contexts, it may be useful to select a *subset* of all valid paths or even just a single path (*e.g.*, shortest path) instead. Doing so, could improve the time complexity of (4) (*e.g.*, selecting shortest paths would yield $O(n^3)$). This path pruning can be done, for instance, by incorporating a sparse matrix in (4). It may also be efficient to apply policy only on edge arbiters of a path, in some contexts (*e.g.*, when enabling traffic measurements).

6 A SERIES OF CASE STUDIES

We now demonstrate the use of our policy-mapping algebras, through seven real SCADA-firewall configuration case studies summarised in Table 1. The data was provided by the authors of (Ranathunga et al., 2015b).

The seven Systems Under Consideration (SUCs), involve various firewall architectures and models. We use them to demonstrate several properties, most notably that the computational complexity of our policy-to-device mapping algorithm is tractable.

An important feature depicted in Table 1 is the *number of security zones* in each network. This number is *small* (*i.e.*, ≤ 21) relative to the maximum (potential) number of hosts per network (*i.e.*, ≤ 67580).

This is to be expected, a zone groups a set of hosts or subnets with identical policies. If every host had a distinct policy then a large number of firewalls would be needed to enforce a real separation between the hosts, making it impractical. By grouping hosts into zones, we reduce policy complexity, so their specification becomes easier and less error-prone.

The typically small zone-count, makes Algorithm 1 (§ 5) computationally feasible. For instance, the worse-case computational complexity involving zones is $O(21^4)$. In comparison, the worse-case computational complexity involving hosts is $O(67580^4)$!

The average time taken by our system to map high-level policy to network devices is 53 seconds per case study. This is when the system is run on a standard desktop computer (*e.g.*, Intel Core CPU 2.7-GHz computer with 8GB of RAM running Mac OS X).

We identified incorrectly mapped (*i.e.*, assigned) ACL rules in each case study by parsing the firewall configurations as per (Ranathunga et al., 2015b). These errors were then classified into three groups: *incorrect-firewall*, *incorrect-interface* and *incorrect-direction* errors (Table 1). *Incorrect-firewall* errors are ACL rules that are assigned to the wrong firewall to begin with, *i.e.*, the desired traffic filtering could not be achieved by placing the ACL rule in any of that firewall's interfaces. *Incorrect-interface* errors are ACL rules that are assigned to the correct firewall but to the wrong firewall-interface, *i.e.*, the

Table 1: SCADA case study summary adapted from (Ranathunga et al., 2015b) (* includes backup firewalls, # ACL rule allocation error).

SUC	Config. date	Fire-walls*	Zones	Conduits	Max. hosts	ACLs	Average rules per ACL	Incorrect firewall#	Incorrect interface#	Incorrect direction#	Runtime (seconds)
1	Sep 2011	3	7	11	67580	8	237	15	13	19	40
2	Aug 2011	6	21	81	2794	12	16	3	2	5	70
3	Oct 2011	4	10	17	886	8	6	2	1	4	43
4	Mar 2011	3	9	16	2038	3	80	5	12	13	61
5	Apr 2015	3	12	19	2664	12	677	15	8	26	47
6	Apr 2015	3	13	21	3562	8	1034	21	15	19	63
7	Jul 2015	6	15	22	3810	17	724	9	5	17	49

desired traffic filtering could not be achieved by assigning the rule inbound or outbound of that firewall-interface. *Incorrect-direction* errors comprise of ACL rules that are assigned to the correct firewall and firewall-interface, but in the wrong direction (e.g., outbound instead of inbound).

As (Table 1) suggests, on average there were 10 ACL rules allocated to the wrong firewall, 8 rules allocated to the wrong firewall-interface and 15 rules allocated in the wrong interface-direction, per SCADA case study. Hence, the intended security policy was *not correctly implemented* in any of these mission-critical networks! Needless to say, what chance do we have in preventing financial loss or moreover, the loss of human lives when our critical infrastructure have incorrectly deployed security policies to begin with?

We automatically mapped the high-level policy in each case to it's network using our system. The generated policy rules were then checked for incorrect allocations. There were *zero incorrectly allocated policy rules*, when the policy was mapped to the firewalls using our algebras! So, the intended high-level security policies were now correctly deployed to the firewalls. By mapping policy to the correct set of firewalls, we reduce vulnerability of these SCADA networks to cyber attack and prevent catastrophic outcomes that result from such attacks.

7 CONCLUSIONS

There are various obstacles that hinder the precise mapping of policies to network devices. Most prominent is the lack of decoupling between policy and network which makes policy sensitive to network-intricacies and vendor changes. To compound the problem, policies can also have complex semantics including node and link properties.

Our research addresses these challenges and proposes a mathematical foundation for mapping policies to network-devices. We use it to deploy real-world security policies to network firewalls provably correctly,

so, that administrators can be confident of the protection provided by their policies for their networks.

REFERENCES

- Anderson, C. J., Foster, N., Guha, A., Jeannin, J.-B., Kozen, D., Schlesinger, C., and Walker, D. (2014). Netkat: Semantic foundations for networks. *ACM SIGPLAN Notices*, 49(1):113–126.
- Anonymous (2015). ForestFirewalls: Getting firewall configuration right in critical networks, <http://tinyurl.com/jdsfewk>.
- Anonymous (2016). Malachite: Firewall policy comparison, <http://tinyurl.com/o2ke3py>.
- ANSI/ISA-62443-1-1 (2007). Security for industrial automation and control systems part 1-1: Terminology, concepts, and models.
- Avelsgaard, C. (1990). *Foundations for Advanced Mathematics*. Scott, Foresman, Brown Higher Education.
- Bartal, Y., Mayer, A., Nissim, K., and Wool, A. (2004). Firmato: A novel firewall management toolkit. *ACM TOCS*, 22(4):381–420.
- Billingsley, P. (1995). Probability and measure. *A Wiley-Interscience Publication*, Wiley & Sons, New York.
- Byres, E., Karsch, J., and Carter, J. (2005). NISCC good practice guide on firewall deployment for SCADA and process control networks. *NISCC*.
- Cisco Systems (2014). *Cisco Virtual Security Gateway for Nexus 1000V Series Switch Configuration Guide*. Cisco Systems, Inc., 170 West Tasman Drive, San Jose, CA 95134-1706, USA.
- Dynierowicz, S. and Griffin, T. G. (2013). On the forwarding paths produced by internet routing algorithms. In *ICNP*, pages 1–10.
- Foster, N., Freedman, M. J., Harrison, R., Rexford, J., Meola, M. L., and Walker, D. (2010). Frenetic: a high-level language for openflow networks. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*, page 6. ACM.
- Graph Steering Committee (2003). GraphML.
- Guttman, J. D. and Herzog, A. L. (2005). Rigorous automated network security management. *IJIS*, 4:29–48.

- Gutz, S., Story, A., Schlesinger, C., and Foster, N. (2012). Splendid isolation: A slice abstraction for software-defined networks. In *ACM HotSDN*, pages 79–84.
- Harary, F. and Norman, R. Z. (1960). Some properties of line digraphs. *Rendiconti del Circolo Matematico di Palermo*, 9(2):161–168.
- Howe, C. D. (1996). *What's Beyond Firewalls?* Forrester Research, Incorporated.
- Johnson, D. S. (2005). The NP-completeness column. *ACM Transactions on Algorithms*, 1(1):160–176.
- Prakash, C., Lee, J., Turner, Y., Kang, J.-M., Akella, A., Banerjee, S., Clark, C., Ma, Y., Sharma, P., and Zhang, Y. (2015). PGA: Using graphs to express and automatically reconcile network policies. In *ACM SIGCOMM*, pages 29–42.
- Ranathunga, D., Roughan, M., Kernick, P., and Falkner, N. (2015a). Towards standardising firewall reporting. In *1st Workshop on the Security of Cyber Physical Systems (WOS-CPS)*. LNCS.
- Ranathunga, D., Roughan, M., Kernick, P., Falkner, N., and Nguyen, H. (2015b). Identifying the missing aspects of the ANSI/ISA best practices for security policy. In *1st ACM Workshop on Cyber-Physical System Security (CPSS)*, pages 37–48.
- Reich, J., Monsanto, C., Foster, N., Rexford, J., and Walker, D. (2013). Modular SDN programming with pyretic. *Technical Report of USENIX*.
- Soulé, R., Basu, S., Marandi, P. J., Pedone, F., Kleinberg, R., Sirer, E. G., and Foster, N. (2014). Merlin: A language for provisioning network resources. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 213–226.
- Twiddle, K., Dulay, N., Lupu, E., and Sloman, M. (2009). Ponder2: A policy system for autonomous pervasive environments. In *ICAS'09*, pages 330–335.
- Yuan, L., Chen, H., Mai, J., Chuah, C.-N., Su, Z., and Mohapatra, P. (2006). FIREMAN: A toolkit for firewall modeling and analysis. In *IEEE SSP*, pages 15–213.
- Zhao, H., Lobo, J., and Bellovin, S. M. (2008). An algebra for integration and analysis of Ponder2 policies. In *POLICY'08*, pages 74–77.

Appendix

Theorem 4 (A^* calculation). A^* can be calculated using the right iteration algorithm

$$A^{<k+1>} = (A^{<k>}T \cup I)A \text{ where } A^{<0>} = I.$$

T and I are the Zone-transitivity matrix and the multiplicative-identity matrix (of semiring $(S, \cup, \cdot, \hat{0}, \hat{1})$) respectively.

Proof. Let's check that the result holds for $k = 0$ (i.e., valid firewall paths between zones up to $(0 + 1)$ hop).

$$\begin{aligned} LHS &= A^{<0+1>} = A^{<1>} \\ RHS &= (A^{<0>}T \cup I)A = (IT \cup I)A \\ &= (T \cup I)A = IA = A \end{aligned}$$

$A^{<1>} = A$ is true since all valid single-hop firewall paths are represented by A .

Let's assume the result holds when $k = n$, i.e., $A^{<n+1>} = ((A^{<n>}T) \cup I)A$.

So, $A^{<n+1>}$ holds all valid firewall paths between the zones of up to $(n + 1)$ hops. Then $A^{<n+1>}(i, j)$ represents all valid firewall paths of up to $(n + 1)$ hops, between zones i and j (see Figure 6). So, valid firewall paths up to $(n + 2)$ hops from i to k via j are given by

$$(t_{ik})_j = A^{<n+1>}(i, j)A(j, k) \quad (13)$$

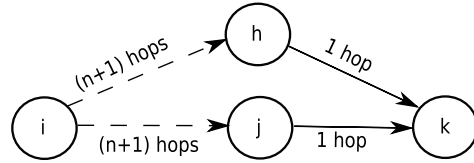


Figure 6: .

But zone j may not be transitive, so we modify the above equation to account for this

$$(t_{ik})_j = A^{<n+1>}(i, j)T(j, j)A(j, k) \quad (14)$$

But, when $i = j$ the above equation yields ϕ , whereas we expect $\{\epsilon\}$ to be the self-firewall path. Again, we update the equation

$$(t_{ik})_j = \left[A^{<n+1>}(i, j)T(j, j) \cup I(i, j) \right] A(j, k) \quad (15)$$

Then, all valid firewall paths up to $(n+2)$ hops from i to k (via j, h, \dots) are given by

$$\begin{aligned} &A^{<n+2>}(i, k) = \\ &\bigcup_{\forall j} \left[A^{<n+1>}(i, j)T(j, j) \cup I(i, j) \right] A(j, k); \forall i, k \quad (16) \end{aligned}$$

We simplify the equation further

$$\begin{aligned} RHS &= \bigcup_{\forall j} \left[A^{<n+1>}(i, j)T(j, j)A(j, k) \right] \cup \\ &\bigcup_{\forall j} I(i, j)A(j, k) \end{aligned}$$

$$\text{But, } A^{<n+1>}(i, j)T(j, j) = \bigcup_{\forall s} \left[A^{<n+1>}(i, s)T(s, j) \right]$$

$$\begin{aligned} \text{So, } &\bigcup_{\forall j} \left[A^{<n+1>}(i, j)T(j, j)A(j, k) \right] \\ &= \bigcup_{\forall j} \left[A^{<n+1>}T \right](i, j)A(j, k) \\ &= \left[A^{<n+1>}TA \right](i, k) \end{aligned}$$

Also, $\bigcup_{\forall j} [I(i, j)A(j, k)] = [IA](i, k)$

Hence, $A^{<n+2>}(i, k) = [A^{<n+1>TA}](i, k) \cup [IA](i, k); \forall i, k$

We can generalise the above to the matrices

$$\begin{aligned} A^{<n+2>} &= A^{<n+1>}TA \cup IA \\ &= (A^{<n+1>}T \cup I)A. \end{aligned}$$

□